TP17: Graphes pondérés, trajets de métro

MP2I Lycée Pierre de Fermat

Algorithme de Floyd-Warshall

On considère des graphes pondérés. Pour un graphe $G = (S, A, w)$, on rappelle que les listes d'adjacence stockent, pour chaque sommet u, une liste de couples (v, p) avec v voisin de u et p poids de l'arête (u, v). Les matrices d'adjacences stockent les coefficients $w(u, v)$ pour $u, v \in S$, et $+\infty$ pour les couples de sommets n'étant pas reliés par une arête. On posera aussi $w(u, u) = 0$ pour tout sommet u.

On signale qu'en C, l'inclusion de la librairie `math.h` permet l'utilisation de la constante `INFINITY`, qui est un flottant se comportant comme $+\infty$. Cette librairie pourra nécessiter de rajouter l'option `-lm` à la fin de la commande de compilation.

Q1. Créer un nouveau fichier C, et y créer un type struct pour les graphes représentés par matrices d'adjacence, que l'on appellera `graphe_t`.

L'algorithme de Floyd-Warshall permet de calculer en $\mathcal{O}(n^3)$ la matrice des distances d'un graphe $G = (S, A, w)$, c'est à dire la donnée de la distance entre toute paire de sommet, pour un graphe **ne contenant pas de cycle négatif**.
On rappelle le principe: pour un graphe $G = (S, A, w)$ avec $S = \{0, \dots, n-1\}$, on calcule pour $i, j \in [\![0, n-1]\!]$ et $k \in [\![0, n]\!]$ la quantité M_{ij}^k = le poids d'un plus court chemin entre i et j ne passant que par des sommets intermédiaires dans $\{0, \dots, k-1\}$. On a la formule de récurrence suivante pour $k \geq 0$:

$$M_{ij}^{k+1} = \min(M_{ij}^k, M_{ik}^k + M_{kj}^k)$$

Cette formule exprime qu'un chemin entre i et j peut soit passer par k, soit pas. On complète avec les conditions initiales:

$$M_{ij}^0 = \begin{cases} +\infty & \text{si } (i,j) \notin A \\ w(i,j) & \text{si } (i,j) \in A \end{cases}$$

Q2. Écrire une fonction `float ** floyd_warshall(graphe_t* g)` qui applique l'algorithme de Floyd-Warshall et renvoie la matrice des distances calculée.

Q3. Tester votre algorithme sur quelques graphes.

Trajet dans le métro

On se propose d'utiliser l'algorithme de Dijkstra pour calculer des plus courts trajets sur la carte du métro parisien.[1]

Cette application nécessitera de manipuler des graphes dont les sommets ne sont pas des entiers, mais des chaînes de caractères. En numérotant les sommets, on peut se ramener à l'utilisation des structures de liste d'adjacence et de matrice d'adjacence que l'on a étudié jusqu'à présent. En stockant la liste des sommets, on peut facilement passer de l'indice d'un sommet à son nom.

Il sera aussi nécessaire de passer rapidement du nom d'un sommet à son indice. Plutôt que de devoir reparcourir la liste des sommets à chaque fois, on décide de stocker les indices dans un **dictionnaire**. Déterminer l'indice d'un sommet donné se fera donc en $\mathcal{O}(1)$ en moyenne, plutôt qu'en $\mathcal{O}(n)$ si l'on opérait plus naïvement.

Le couple de fichiers `stoi.h`/`stoi.c` (**S**tring **TO** **I**nt) contient une implémentation par tables de hachage de dictionnaires prenant comme clés des chaînes de caractère et comme valeurs des entiers. On utilisera le type suivant pour les graphes:

```c
// liste chaînée de couples (voisin, poids de l'arête)
struct adj {
  int v; // indice du voisin;
  float w; //poids de l'arête vers v
  struct adj* suiv; // maillon suivant
}
typedef struct adj adj_t;

struct graphe {
  int n; // nombre de sommets;
  char** sommets; // sommets[i] contient le nom du sommet d'indice i
  adj_t** voisins; // voisins[i] est la liste des voisins du sommet d'indice i
  stoi_t* indice; // dictionnaire: indice[u] contient l'indice du sommet de nom u
};
typedef struct graphe graphe_t;
```

Dans l'archive du TP figure:

- Le couple `stoi.h`/`stoi.c`, dont le code est complet et utilisable. La documentation figure sur la dernière page du TP.
- Une couple `graphe.h`/`graphe.c` contenant la définition de la structure ci-dessus, ainsi que deux fonctions `afficher` et `est_arete` vous montrant deux exemples basiques de manipulation de cette structure.
- Un fichier `test.c` dont le main permet de créer le graphe G_0 ci-dessous. Le code ne compile par pour le moment car il fait référence aux fonctions que vous allez implémenter dans les premières questions.

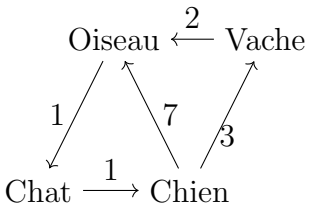

Figure 1: Graphe G_0

[1] Le métro de Toulouse est plus local mais moins intéressant d'un point de vue topologie de graphe...

Pour commencer, implémentons des opérations de base sur les graphes. Les prochaines fonctions doivent être mises dans `graphe.h/graphe.c`

Q4. Regarder dans le fichier `stoi.h`, et expliquer à quoi servent les lignes 1, 2 et 32. Implémenter le même mécanisme pour `graphe.h`.

Q5. Implémenter une fonction `graphe_t* graphe_vide(char** sommets, int n)` créant un graphe à n sommets (donnés par le tableau `sommets`), sans arête.
Implémenter une fonction `void ajouter_arete(graphe_t* g, char* u, char* v, float w)` ajoutant une arête (u, v) dans le graphe g, de poids w.
Compiler `test.c` pour vérifier que tout fonctionne bien

Q6. Dessiner schématiquement l'état de la mémoire après la création du graphe G_0.

Q7. Écrire une fonction `void graphe_free(graphe_t* g)` libérant toute la mémoire allouée pour un graphe.

Algorithme de Dijkstra

Q8. Créez deux fichiers `dijkstra.h/dijkstra.c`. Vous y ajouterez les différentes fonctions de cette section. Ajoutez les gardes d'inclusion dans `dijkstra.h`.

Q9. Écrire une fonction `float longueur(graphe_t* g, char** chemin, int n)` prenant en entrée un graphe `g` et un tableau `chemin` de `n` sommets, permettant de calculer la longueur totale du chemin dans le graphe.

On rappelle le pseudo-code de l'algorithme de Dijkstra:

Algorithme 1 : PCC: Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré avec $s = \{0, \dots, n-1\}$, $s \in S$
Sortie(s) : **d** tableau des distances depuis s, et **Pred** tableau des prédecesseurs

```
1  d ← tableau de taille n initialisé à [+∞, ..., +∞];
2  Pred ← tableau de taille n initialisé à [−1, ..., −1];
3  d[s] = 0;
4  Q ← ensemble contenant chaque élément de S;
5  tant que Q non vide faire
6      u ← extraire sommet de Q avec d[u] minimal;
7      pour v voisin de u faire
8          si d[u] + w(u, v) < d[v] alors
9              d[v] ← d[u] + w(u, v);
10             Pred[v] ← u;
11 retourner d, Pred
```

Nous allons implémenter la structure Q (une file de priorité) de manière naïve, en utilisant un tableau de booléen indiquant, pour chaque sommet, s'il est encore présent dans Q ou non. Initialement, toutes les cases sont vraies pour indiquer qu'aucun sommet n'a été choisi.

Q10. Écrire une fonction `int extraire_min(float* d, bool* q, int n)` cherchant un entier $i \in [\![0, n-1]\!]$ pour lequel $q[i] = 1$ et tel que $d[i]$ est minimal. Cette fonction renverra l'entier en question et mettra la case correspondante dans q à 0. Si q est vide (i.e. n'a que des éléments à faux), la fonction renverra -1.

Q11. Écrire une fonction `int* dijkstra(graphe_t* g, char* s)` qui lance l'algorithme de Dijkstra depuis le sommet s et renvoie l'arborescence du parcours sous la forme du tableau des prédécesseurs.

Q12. En déduire une fonction `char** pcc(graphe_t g, char* s, char* t)` qui renvoie un plus court chemin de s à t sous la forme d'un tableau. s et t seront respectivement le premier élément et le dernier élément du tableau, ce qui permettra de le manipuler sans connaître sa taille. Il pourra être plus simple de reconstruire d'abord le chemin à l'envers, puis de le renverser.

Q13. Tester votre fonction sur quelques graphes.

Lecture de fichiers On stocke les graphes orientés dans des fichiers texte. Pour stocker un graphe $G = (S, A, w)$ pondéré à n sommets et m arêtes, on écrira dans un fichier:

- Sur la première ligne, n et m;
- Sur les n lignes suivantes, les noms des différents sommets, qui peuvent comporter des espaces, mais pas de signe "$";
- Sur les m lignes suivantes, des triplets de la forme `u$v$d` où (u, v) est une arête et $d = w(u, v)$. Il n'y a pas d'espace autour des dollars.

Vous trouverez dans l'archive du TP un couple de fichiers `parser.h/.c` contenant notamment une fonction `graphe_t* lire_graphe(char* filename)` permettant de lire un graphe sous ce format. L'archive contient aussi un fichier `bebe_graphe.txt` représentant un graphe sous ce format.

Q14. Tester la fonction `lire_graphe` sur `bebe_graphe.txt`.

Métro L'archive du TP contient un dossier "metro", contenant lui même 14 fichiers "`ligne_1.txt`", "`ligne_2.txt`", ..., "`ligne_14.txt`" contenant les informations des 14 lignes de métro parisiennes (lignes bis exclues) sous la forme de graphes. On souhaite pour commencer réunir ces informations dans un unique graphe.

Rappel: la fonction `sprintf` sert à écrire dans un string en utilisant les mêmes mécanismes que `printf`. Par exemple, `sprintf(s, "ligne_%d.txt", 12)` va remplacer s par `ligne_12.txt`. Il faut cependant avoir réservé assez d'espace dans s.

Q15. Écrire une fonction `graphe_t* fusion_graphes(graphe_t** L, int p)` qui prend en entrée une liste L de p graphes, et renvoie un graphe dont:

- L'ensemble des sommets est l'union de l'ensemble des sommets des graphes de L;
- L'ensemble des arêtes est l'union de l'ensemble des arêtes des graphes de L.

Ainsi, une station présente sur plusieurs lignes sera représentée par un unique sommet. **Indications:** on pourra utiliser un dictionnaire pour assigner à chaque sommet un numéro unique: 0, puis 1, puis 2, et ainsi de suite, afin de pouvoir compter facilement le nombre de sommets distincts du graphe résultant.

Q16. Sur le graphe construit, chercher le plus court chemin entre `Porte d'Auteuil` et `Pyramides`. Vous pouvez vérifier la cohérence de vos résultats sur un plan du métro en ligne. Combien de changements y a t-il ?

On veut maintenant modéliser le fait qu'un changement n'est pas instantané, et que les jours fériés et les week-ends, un changement peut même être très coûteux. Pour cela, on propose une deuxième manière de fusionner les graphes:

Étant donnés p graphes $G_1, \dots G_p$ et $t \geq 0$ un temps d'attente, la fusion de $G_1, \dots G_p$ avec temps de correspondance t est le graphe $G = \mathbf{Corresp}^t(G_1, \dots G_p)$ obtenu en mettant côte à côte les graphes $G_1, \dots G_p$ sans aucune arête inter-graphe, puis en connectant tous les sommets de même nom, par une arête de poids t.

Ainsi, un sommet u présents dans les graphes G_i, G_j, G_k donnera lieu à trois sommets u_i, u_j, u_k formant une clique où toutes les arêtes sont de poids t.

Q17. Dessiner trois graphes F, G et H ayant quelques noms de sommets en commun, puis dessiner le graphe $\mathbf{Corresp}^t(F, G, H)$.

Afin de représenter cet opérateur dans notre programme, il faut donner un nom aux nouveaux sommets créés. Pour la ligne numéro i, une station u aura dans le graphe des correspondances le nom $u\#i$. Par exemple, la station `Gare Montparnasse` est sur les lignes 4, 6, 12 et 13. Dans le graphe des correspondances elle sera alors remplacée par 4 sommets: `Gare Montparnasse#4`, ..., `Gare Montparnasse#13`.

Q18. Écrire une fonction `graphe_t* correspondances(graphe_t** L, int p, float t)` qui prend en entrée une liste $L = [G_1, \dots, G_p]$ de p graphes ainsi qu'un temps d'attente t, et crée le graphe des correspondances $G = \mathbf{Corresp}^t(G_1, \dots, G_p)$.

Nous avons maintenant tous les outils nécessaires pour écrire un programme donnant des trajets de métro efficaces. Le but des dernières questions est d'obtenir un exécutable pouvant s'utiliser comme suit:

```
./trajet "Porte d'Auteuil" "Pyramides" 4

Trajet de Porte d'Auteuil à Pyramides (Temps total: 24 minutes):

1) Ligne 9 de Porte d'Auteuil jusqu'à Chaussée d'Antin La Fayette (17 minutes)

2) Changement à Chaussée d'Antin La Fayette (4 minutes)

3) ligne 7 de Chaussée d'Antin La Fayette jusqu'à Pyramides (3 minutes)
```

Le dernier paramètre donné est le temps de correspondance moyen attendu pour une station.

Q19. Commencez par simplement afficher la liste des stations à prendre, en énumérant les sommets d'un plus court chemin, ainsi que le temps total de trajet.

Q20. Tester votre programme en vérifiant les résultats sur la carte du métro, et vérifiez qu'en augmentant le temps d'attente aux stations, votre programme privilégie les trajets ayant peu de changements.

Q21. Améliorez votre programme pour qu'il affiche seulement les lignes à prendre et les changements à effectuer, ainsi que le temps passé sur chaque ligne, plutôt que la liste des stations.

Annexe: Documentation

Dictionnaires string to int

On rappelle que les chaînes de caractères sont des **pointeurs**. Dans la structure de dictionnaire fournie, les chaînes données comme clés ne sont pas recopiées en mémoire, seule leur adresse est stockée. La fonction `void stoi_free(stoi_t* T)` ne libère pas la mémoire allouée aux clés de T.

```
1  /* Crée une table de hachage vide */
2  stoi_t* stoi_vide();
3
4  /* Renvoie true si `clef` est une clef de t, false sinon */
5  bool stoi_mem(stoi_t* t, char* clef);
6
7  /* Renvoie la valeur associée à `clef` dans t. Si
8     t ne contient pas `clef`, affiche un message d'erreur
9     et arrête l'exécution. */
10 int stoi_get(stoi_t* t, char* clef);
11
12 /* Assigne à `clef` la valeur `valeur` dans t. Si
13    `clef` est déjà présente, modifie la valeur précédente */
14 void stoi_set(stoi_t* t, char* clef, int valeur);
15
16 /* Libère l'espace alloué pour la table t.
17    Attention: ne libère pas les clés.*/
18 void stoi_free(stoi_t* t);
```